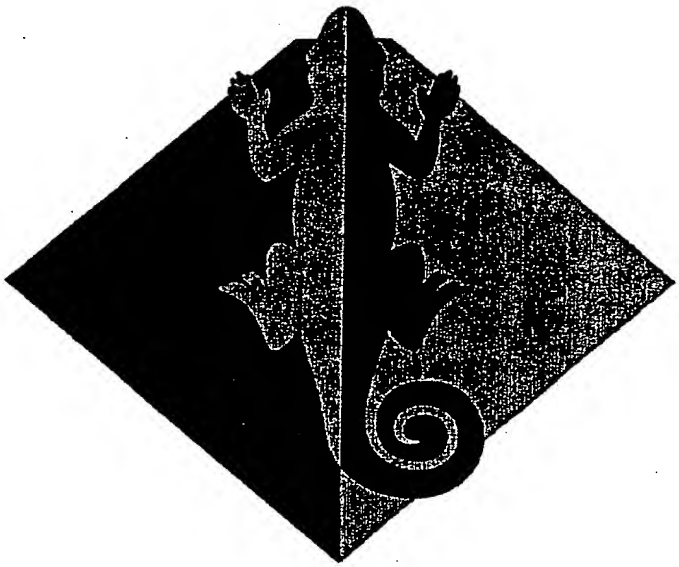


CHAMELEON

SYSTEMS, INC.

Design and Implementation of a Parallel Viterbi Decoder

by Dan Pugh

The following are trade names, trademarks, or registered trademarks of their respective companies:

Version History

Date	Version	Description
11/17/99	_0.1	Initial version. Author(s): Dan Pugh

For additional copies of this document, please contact:

Chameleon Systems, Inc.
1195 W. Fremont Avenue
Sunnyvale, CA 94087
Telephone: 408-730-3300
Facsimile: 408-730-3303

1.0 Overview

This paper outlines the design and implementation of a parallel Viterbi decoder that decodes the convolutional codes such as those used in IS-95 and IS-2000 CDMA systems. This design maintains the decoder functionality, but it deviates from the traditional Viterbi decoders in several areas:

- The structure of the generator polynomials in the convolutional code has been exploited in order to optimize the decoding process for parallel implementations.
- The individual branch metrics, although unique, are related. Some of the branch metrics must be calculated while others may be implied. By leveraging the structure of the generator polynomials in the convolutional encoder, the branch metric calculations are simplified.
- The path metric update calculations are performed in parallel to reduce path metric update times.
- A new memory structure has been added to the decoder that eliminates the need to perform a serial trace back through the decoder states in order to obtain the best estimate of the transmitted data. This new memory structure eliminates the need to store traceback pointers for each of the individual states. Rather than storing traceback pointers for each state, this design stores the maximum likelihood transmitted path into each of the states.
- In order to further reduce the overall computational time of the decoding process, the branch metric calculations and the path metric operations are pipelined so that they may operate in parallel. While the path metrics are being updated for one symbol, the decoder simultaneously computes the branch metrics for the next.

1.1 Terminology

The following symbols will be used is the description of the Viterbi decoder:

K Constraint Length, equal to one more than the number of state variables

N_S Number of states.

MLD Maximum Likelihood Decision

ACS Add, Compare, Select.

BM Branch Metrics

PM Path Metric

2.0 Theory of Operation

2.1 Convolutional Encoder

This paper outlines the implementation of a constraint length 9 (K = 9), rate 1/4 convolutional encoder. IS-2000 defines rate 1/2, rate 1/3, rate 1/4 convolutional encoders. As an example the K = 9, rate 1/4 convolutional encoder of the IS-2000 system will be used.

The generator functions for the rate 1/4 code are defined as:

$G_0(D) = 1 + D + D^2 + D^3 + D^4 + D^6 + D^8$ $\quad G_0 = 765$ (octal)

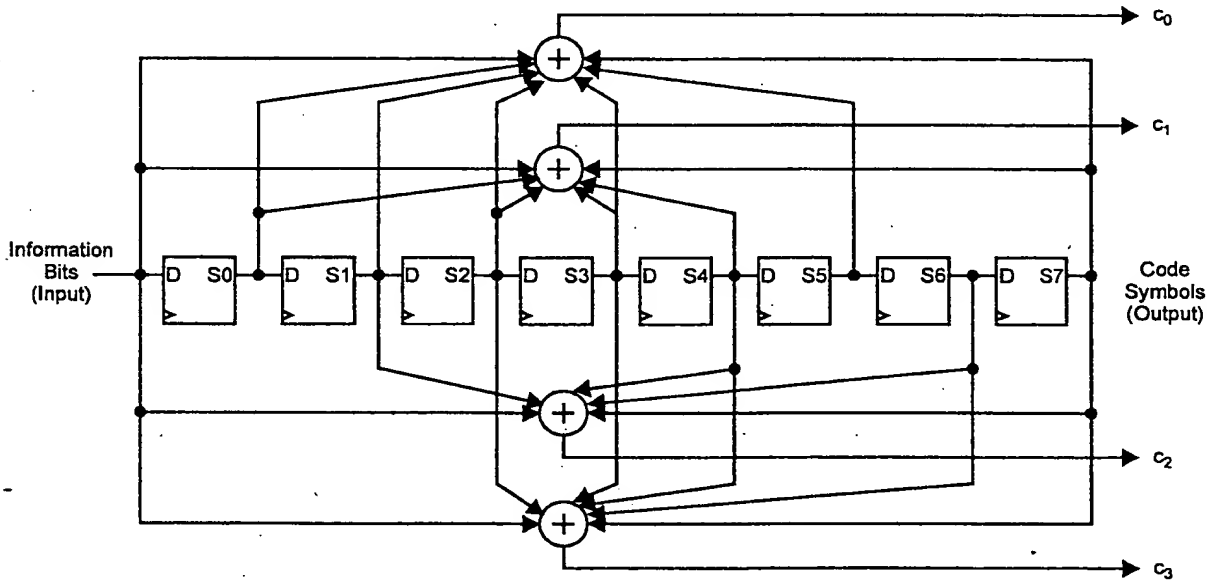
$G_1(D) = 1 + D + D^3 + D^4 + D^5 + D^8$ $\quad G_1 = 671$ (octal)

$G_2(D) = 1 + D^2 + D^5 + D^7 + D^8$ $\quad G_2 = 513$ (octal)

$G_3(D) = 1 + D^3 + D^4 + D^5 + D^7 + D^8$ $\quad G_3 = 473$ (octal)

The state of the convolutional encoder is initialized to state zero before the first output symbol is transmitted. Each information bit is encoded into four output code symbols, c_0, c_1, c_2 and c_3, are transmitted independently (BPSK for the example of IS-2000) starting with c_0. If c_n=0, maps to the -1 symbol, while c_n=1 maps to the +1 symbol. Figure 2-1 shows the structure of the K=9, rate 1/4 convolutional encoder.

Figure 2-1. K =9, Rate 1/4 Convolutional Encoder



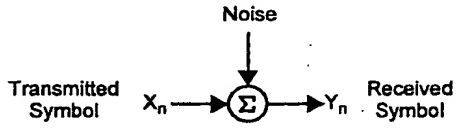
2.1.1 Block Convolutional Encoder

In many transmission systems, the transmitted symbols are divided into fixed block sizes. If the system has extra bandwidth, the decoder design may be simplified by appending extra zeroes (tail bits) to the end of the data bits in the transmitted block. If enough tail bits are transmitted, the convolutional encoder will be forced back to the zero state at the end of the block. This number of bits is defined by $N_{tailbits} = \log_2(K - 1)$, where K is the constraint length of the convolutional code. If the convolutional encoder is forced to the zero state at the end of the block, the decoder does not need to search for the minimum energy state at the end of a block, it is already known to be zero. The example of this paper shows the implementation of a block convolutional encoder/decoder, but the core of the design may be used in either a streaming or fixed block mode.

2.2 Channel Noise Model

The noise introduced by the transmission mediums, such as line-of-sight terrestrial channels can be accurately modeled by the additive white Gaussian noise (AWGN) model. The demodulated symbol, Y_n, may be represented by a random noise component added to the original transmitted signal, X_n.

Figure 2-2. AWGN Model



The probability density function of the Gaussian noise with nonzero mean and variance σ for a single dimensional signal is given by:

$$p(y) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{(Y_n - Y_0)^2}{2\sigma^2}}$$

where σ is the root-mean-square value of the noise voltage and Y_0 is the mean or dc value. We see that the probability density function is a function of $(Y_n - Y_0)2$. This squared relationship is the basis for using the square of the Euclidian distance as the branch metric function. The other terms in the probability density function may be ignored for use in the branch metric calculation because they will be common to all of the metrics and only introduce a scaling or DC offset effect.

2.3 Viterbi Decoder

The Viterbi algorithm, originally published in 1967 is a maximum likelihood technique. Convolutional encoding with Viterbi decoding has become one of the most widely used forward-error-correction (FEC) techniques. The following sections give a general description of the Viterbi decoding algorithm. A detailed explanation of the Viterbi decoding algorithm is beyond the scope of this paper, but may be found in many communications texts. The Viterbi decoder may be decomposed into several primitive elements:

- Read Input Symbols
- Compute Branch Metrics
- Update Path Metrics
- Determine maximum-likelihood path through the trellis (traceback)

2.3.1 Read Input Symbols

The input to the Viterbi decoder consists of either single or multidimensional symbols. For the example of this paper, we assume a rate 1/4 convolutional encoder whose outputs are transmitted independently over a BPSK channel. The four outputs of the convolutional encoder compose a single four-dimensional orthogonal signal. Because the four outputs are transmitted independently, they are orthogonal to each other and the AWGN channel noise model may be applied independently to each of the four outputs of the composite symbol.

In some systems such as IS-95 or IS-2000, symbols are replicated and transmitted redundantly. The design example of this paper assumes that the symbol replication method is not employed. If a communications system uses this method, the composite components of the symbols should be combined before they are forwarded to the branch metric circuit. The core of the model shown here will work with or without symbol replication, only the input circuit will be effected.

2.3.2 Branch Metric Computation

The branch metric is used to determine the maximum-likelihood probability that the received symbol is the same as the transmitted symbol for a given state of the convolutional encoder. Since the state of the encoder is not known for every received symbol, branch metrics need to be calculated for each of the 2^{K-1} states.

For the symbol received, Y_n, at time n, the branch metric for state k is defined as the squared Euclidean distance between Y_n and the expected signal, X_{km}. For BPSK, X_{km} (m = encoded data bit = 0,1) is assumed to take the values of either -1 or +1, respectively. For a single dimensioned BPSK symbol, the squared distance calculation for the branch metric at time n for information bit m, is given by:

$$BM_n(k,m) = (Y_n - X_{km})^2 = Y_n^2 - 2Y_nX_{km} + X_{km}^2$$

$$BM_n(k,0) = (Y_n - X_0)^2 = Y_n^2 + 2Y_n + 1$$

$$BM_n(k,1) = (Y_n - X_1)^2 = Y_n^2 - 2Y_n + 1$$

One should note that the Y_n^2 term is common to both of the branch metrics for a given received symbol and the X_m^2 term is always equal to 1. Since the $Y_n^2 + X_{km}^2$ term can be considered a dc offset, and the -2 term is a constant scaling factor, the branch metric is then reduced to:

$$BM_n(k,m) = -Y_n X_{km}$$

$$BM_n(k,0) = +Y_n,$$

$$BM_n(k,1) = -Y_n$$

For the example of the rate 1/4 encoder defined in Figure 2-1, the four encoded symbol bits, $c_0 - c_3$, are mapped into four separate BPSK symbols that are transmitted independently. Because the BPSK symbols are independent, there noise components are also independent and the AWGN model for the channel is still valid. When four independent BPSK symbols are combined into a single four-dimensional symbol for the Viterbi decoder, the branch metric becomes:

$$BM_n(k,m) = \Sigma[(Y_{ni} - X_{mi})^2], \{i=0, 1, 2, 3\} = (Y_{n0} - X_{m0})^2 + (Y_{n1} - X_{m1})^2 + (Y_{n2} - X_{m2})^2 + (Y_{n3} - X_{m3})^2$$

which reduces to: $BM_n(k,m) = -Y_{n0}X_{m0} + -Y_{n1}X_{m1} + -Y_{n2}X_{m2} + -Y_{n3}X_{m3}$

Since X_{mi} can only take the values of -1 or +1, the branch metric is reduced to the sum or difference of each of the four received composite symbols $Y_{n0} - Y_{n3}$.

2.3.3 Path Metric Computation

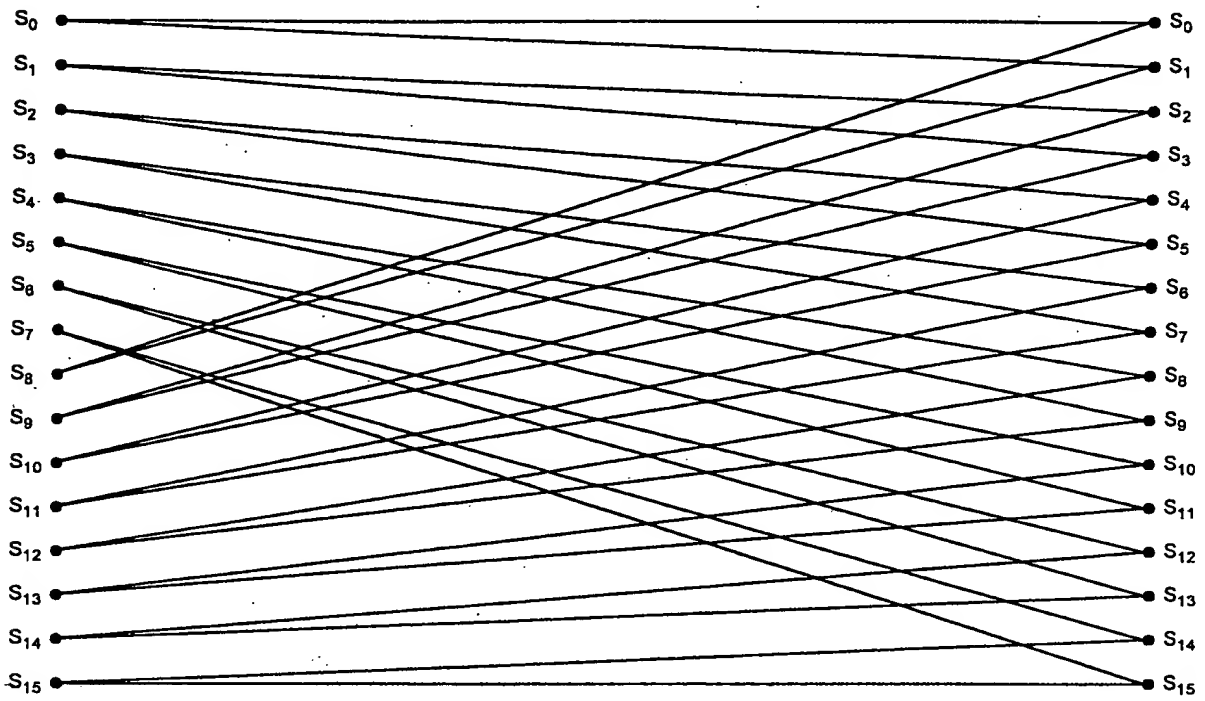
To understand how the convolutional encoder changes state from one symbol interval to the next, we may look at the convolutional encoder in Figure 2-1, where the state of the encoder is defined by the bits in registers S0 through S7. Upon the rising edge of the symbol clock, a new information bit is shifted into state variable S0, and state variables S1 through S6 are shifted into state variables S2 through S7. This translates into two possible state transitions from each state k, depending on the value of the information bit i:

If i=0, nextstate = 2k

If i=1, nextstate = 2k + 1

To show the 256-state convolutional encoder state transition graph of the encoder outlined in this design example is impractical. For illustrative purposes, Figure 2-3 shows the state transition graph of a similarly structured sixteen-state convolutional encoder.

Figure 2-3. 16-State Convolutional Encoder State Transition Graph



For each received symbol, the path metric is calculated for each of the 2^{K-1} states to determine the maximum-likelihood probability that the convolutional encoder is in a specific state. This is done by calculating the minimum energy path into each state and updating the metric. For each state there are two possible paths into that state.

To define the input paths into a given state, the floor function, $\lfloor x \rfloor$, must first be defined. The floor of a positive rational number is the closest whole integer less than or equal to the number. For example $\lfloor 2.7 \rfloor = 2$ and $\lfloor 3 \rfloor = 3$. For each state k there are two possible input paths:

state k previous state path 1: state $\lfloor k/2 \rfloor$

state k previous state path 2: state $(\lfloor k/2 \rfloor + 2^{K-2}) \bmod 2^{K-1}$

The circuit updates the path metrics by calculating the path metrics for both of the possible paths into a given state and selecting the one with the lower value. Lower path metric values are associated with

a received symbol being [redacted]r to the predicted transmitted symbol. The pat[redacted]tric PM(k), for state k is defined as:

New path metric = PM(k) = MINIMUM(A, B), with

A = path metric path 1 into state k:$(PM(\lfloor k/2 \rfloor + BM_n(\lfloor k/2 \rfloor, m))$

B = path metric path 2 into state k:$(PM((\lfloor k/2 \rfloor + 2^{K-2}) \bmod 2^{K-1}) + BM_n((\lfloor k/2 \rfloor + 2^{K-2}) \bmod 2^{K-1}), m)$

where m=0 for k=even and m=1 for k=odd

The path metric update circuit does an add, compare, select (ACS) function of the two previous path metrics.

2.3.4 Determine maximum-likelihood path through the trellis (traceback)

This decoder design uses a new technique to pickup the maximum-likelihood path through the convolutional encoder's trellis. Before the new algorithm is discussed, the traditional method is reviewed for comparison.

2.3.4.1 Traditional Viterbi Traceback Algorithm

The traditional Viterbi decoding algorithm keeps a history of state pointers as it processes path metrics and determines the maximum likelihood state transition into each state. After the $L_{BLOCKLENGTH}^{th}$ symbol is received and the path metrics are updated for each of the states, the Viterbi algorithm finds the state with the minimum path metric. The traceback pointer associated with this minimum path metric is examined to determine which state was the maximum likelihood state at the previous symbol interval. If the traceback pointer is 0, then the maximum likelihood state into state n is state $\lfloor n/2 \rfloor$. If the traceback pointer is 1, then the maximum likelihood state into state n is state $(\lfloor n/2 \rfloor + 128)$. The Viterbi algorithm then examines the traceback pointer for state $\lfloor n/2 \rfloor$ or $(\lfloor n/2 \rfloor + 128)$, at time k-1, to determine if the value of the traceback pointer for state n at time k is 0 or 1 respectively. This traceback procedure is repeated to traverse back through the trellis at least $L_{TRACEBACK}$ (at least three or four constraint lengths) units in time in order to ensure convergence of the traceback path with the optimal path. At this point a single data bit is decoded and sent to the output circuit. The algorithm may be altered where several symbols are picked up, but the serial process of the reverse traversal through the trellis is still required.

To aide block Viterbi decoding, L_{TAIL} zeroes (tail bits) may be transmitted at the end of the block of L_{DATA} data bits to guarantee that the convolutional encoder returns to state zero at the end of the block of length $L_{DATA} + L_{TAIL}$. While transmitting the tail bits eliminates having to search for the minimum energy state at the end of the block, the trellis still must be traversed backwards in order to determine the maximum likelihood transmitted bit sequence. Since tracing back from one state to the previous depends on the present state, this process must be repeated $L_{DATA} + L_{TAIL}$ times until all of the transmitted bits are

determined. Even for the simpler case of block decoding, the Viterbi algorithm's traceback operation is a serial process and does not lend itself to a faster, parallel implementation.

2.3.4.2 Improved Maximum-Likelihood Path Detection Method

The traditional Viterbi traceback algorithm discussed above is a serial process and cannot be implemented in a highly parallel fashion. Even when the branch metric and path metric circuits are improved with parallel implementations, the traceback process remains a serial process. The traditional Viterbi decoding method updates the path metrics to find the maximum-likelihood state of the decoder and then traces back a single path through the trellis. This path corresponds to a specific transmitted bit pattern. The procedure for decoding points in the algorithm presented here is quite different. For simplicity, only the procedure for block decoding will be described. Similar operations can be performed to implement a continuous stream decoder.

Rather than storing a single traceback pointer for each of the last $L_{DATA} + L_{TAIL}$ units in time for each of the states, the data bits for the maximum likelihood transmitted sequence for the last W_{TP} (Traceback Path Memory width) data bits are stored. The serial traceback procedure may be eliminated if a new data structure is added to the output of the add, compare, select (ACS) block. If instead of storing a single pointer in the traceback memory for each of the states, the entire optimal path leading up to that path is stored in the Optimal Path (OP) memory. As each symbol is received by the decoder, the OP circuit updates the optimal paths for each of the 2^{K-1} states. The OP circuit reads the optimal path of the state indicated by the traceback pointer generated by the ACS block. The bit corresponding to a transition from the best previous state to the given state is appended to the OP memory contents of the selected optimal path and then stored back in the OP memory.

In a parallel implementation the OP circuit prefetches the previous optimal paths of the two states that precedes the state to be updated. The OP circuit uses the traceback pointer of the ACS circuit to select which of the two optimal paths is selected. Note that the two previous state OP memory addresses are the same as the previous state path metric memory addresses. The OP circuit may be implemented in parallel with the ACS block in order to share address generators for the OP and PM memories.

As each symbol is received, a new bit is appended to the OP memory. The OP memory contents grow with the addition of each new bit in the optimal path for each state. Since it is usually impractical to perform a read/modify/write operation on the entire history of an optimal path, the OP memory must be divided into workable blocks.

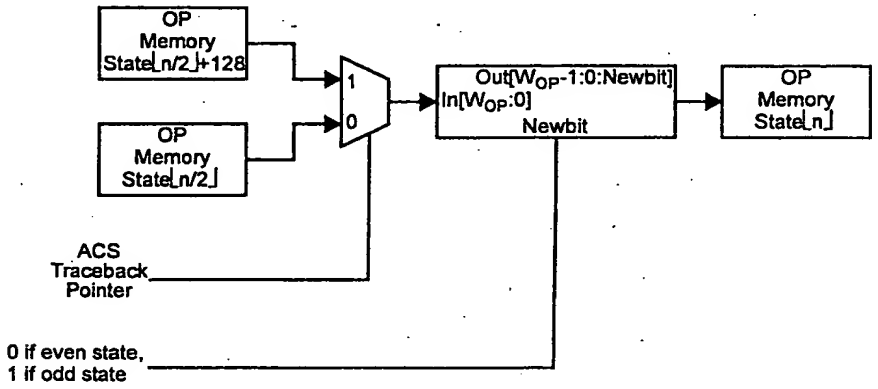
2.3.4.2.1 Optimal Path (OP) Memory Update Algorithm

The ceiling operator, $\lceil x \rceil$, is defined as the closest integer greater than or equal to the argument x. For example $\lceil 2.2 \rceil = 3$ and $\lceil 5 \rceil = 5$.

The OP memory may be divided into practical word widths. Let W_{OP} be defined as the bit width of the OP memory. If a block decoder of $L_{BLOCKLENGTH}$ bits is implemented, there will need to be $L_{OPBLOCKS} = \lceil L_{BLOCKLENGTH} / W_{OP} \rceil$ blocks, of memory of length 2^{K-1}. The following C code updates the OP memory contents.

```
void Update_OP_Memory (int TracebackPointer)
#define NUMSTATES 256
#define MEMORYWIDTH 32
int NumBitsShifted = 0;
int OP_Memory_Block_Index = 0;
for (ReceivedSymbol=0;ReceivedSymbol<BlockSize;ReceivedSymbol++)
begin
  NumBitsShifted = NumBitsShifted + 1;
  if (NumBitsShifted == MEMORYWIDTH)
    begin
      OP_Memory_Block_Index = OP_Memory_Block_Index + 1;
      NumBitsShifted = 0;
    end
  int state;
  int BestPath;
  begin
    for (state=0;state<NumStates;state++)
    begin
      if TracebackPointer
        BestPath = OP_Memory[state/2];
      else
        BestPath = OP_Memory[(state/2)+ (NUMSTATES / 2)]
      if ((state % 2) == 0)/* is the state even */
        OP_Memory[state] = BestPath << 1;
      else /* the state is odd */
        OP_Memory[state] = BestPath << 1 | 1; /*append 1 to the shifted word */
    end;
  end;
end;
```

Figure 2-4. Optimal Path (OP) Memory Update Algorithm Block Diagram



2.3.4.2.2 Optimal Path Assembly Algorithm

In order to simplify the explanation of the Optimal Path Assembly Algorithm, a block convolutional encoder is assumed. The algorithm will also work with a standard, streaming mode convolutional encoder with a few minor modifications to the block code version.

The Optimal Path Memory Update algorithm stores the transmitted bits of the optimal path leading into each state. If the path is longer than the width, W_{OP} of the OP memory, $L_{OPBLOCKS}$ words will be required to store the optimal path of each state. After all of the symbols in a block have been received and the path metrics and OP memories have been updated, the maximum-likelihood decoded bits may be read from the OP memories.

In order to better understand state transitions, one should look at the structure of the convolutional encoder shown in Figure 2-1. Figure 2-1 shows that the information bits get shifted into the state register bits from the least significant bit up to the most significant bit. As long as at least K-1 bits have been processed through the encoder, the state register contains the last K-1 information bits.

The decoded bits are read out of the OP memory starting with the most recently used OP block, which is denoted $OPBLOCK_{LAST}$. The first OP word is read from the OP memory location, $OPBLOCK_{LAST}$(min_pathmetric), corresponding to the state with the minimum path metric. If the convolutional encoder utilizes tail bits, as is the case with IS-95 and IS-2000 systems, the minimum energy final state is guaranteed to be state zero.

If $L_{BLOCKLENGTH}$ is not an integer multiple of W_{OP}, not all of the bits of the word in OP block $OPBLOCK_{LAST}$ are valid. If the last location is not completely filled, only the $L_{BLOCKLENGTH}$ % W_{OP} least significant bits will be valid, where % is the modulus operator. The modulus function returns the remainder when one integer is divided by another. To pickup the next W_{OP} maximum-likelihood decoded bits, we need

to know which state was previous state to state $OPBLOCK_{LAST}$(min_pathmetric). In order to better understand state transitions, one should look at the structure of the convolutional encoder shown in Figure 2-1. Figure 2-1 shows that the information bits get shifted into the state register bits from the least significant bit up to the most significant bit. As long as at least K-1 bits have been processed through the encoder, the state register contains the last K-1 information bits. For a 2^{K-1} state convolutional encoder, the K-1 state bits correspond to the last K-1 transmitted information bits. We know that $OPBLOCK_{LAST}$(min_pathmetric) is the final state of the memory, and that the bottom K-1 bits of $OPBLOCK_{LAST}$(min_pathmetric) are also the same as the K-1 bit number of the index of $OPBLOCK_{LAST}$(min_pathmetric). Therefore the previous state into $OPBLOCK_{LAST}$(min_pathmetric) will be $OPBLOCK_{LAST-1}$(($OPBLOCK_{LAST}$(min_pathmetric))[K:1] >> 1). We repeat this process again to get the next W_{OP} previous bits in the maximum-likelihood path. The maximum-likelihood previous state into $OPBLOCK_{LAST-1}$($OPBLOCK_{LAST}$(bestpath)) will be $OPBLOCK_{LAST-2}$(($OPBLOCK_{LAST-1}$(min_pathmetric))[K:1] >> 1). This process is repeated until all of the OP memory blocks have been read.

The following C subroutine describes the optimal path assembly algorithm:

```
procedure Read_Optimal_Path ()
   #define OPMAXBLOCK 256
   #define NUMSTATES 256
   #define OPINDEXMASK 0xFF
   int OP_Mem[OPMAXBLOCK][NUMSTATES];
   int Block_Index;
   int OP_Block;
   int OutWord[OPMAXBLOCK];
   begin
      Block_Index = Find_Min_State(last_state)
      /* MinLastState = 0 for block encoders with tail bits */
      for (OP_Block=OPMAXBLOCK; OP_Block=0; OP_Block--)
        begin
          OutWord[OP_Block] = OP_Mem[OP_Block,Block_Index]
          Block_Index = OP_Mem[OP_Block,Block_Index] << 1 ^ | OPINDEXMASK;
        end
   end;
```

3.0 Viterbi Decoder Implementation

The following sections detail the design of a Viterbi decoder with the modified traceback algorithm in the Chameleon Systems CS2112 programmable system-on-a-chip (PSOC) processor. These sections serve to both detail the design of the decoder as well as educate the user as to how to map large designs onto the CS2000 series of PSOC processors. The reader should refer to the Chameleon Systems CS2000 databook for a detailed description of the CS2112 PSOC.

3.1 Decoder Design Assumptions

- The design is being mapped to a Chameleon Systems CS2112 PSOC.
- A primary design goal is to maximize the performance of the decoder
- A highly pipelined architecture is required to achieve maximum performance
- The blocksize is defined such that no off-chip memory is required
- The design is for the K=9, rate 1/4 convolutional encoder of Figure 2-1
- The four symbol components of each encoder symbol is orthogonal
- The transmission channel can be modeled with an AWGN noise model
- The generator polynomials are defined in Section 2.1
- The blocksize, $L_{BLOCKLENGTH}$, is 256 symbols
- The encoder sends 8 tail bits at the end of the block
-

3.2 Decoder Design Partitioning

Since the primary design goal is to maximize performance, the decoder will be designed such that it may draw from all of the CS2112 resources. The design will thus be mapped into all four slices of the part.

3.2.1 Decoder Datapath Partitioning

The decoder design is a memory intensive design. Storage is required for the branch metrics, path metrics, and the optimal path memory.

3.2.2 Decoder Control Partitioning

hierarchical control partitioning

distributed control

redundant control required if in different slices to minimize global interconnect

3.2.3 Decoder Routing Issues

3.3 Decoder Circuits

3.3.1 Decoder Input Circuit

number of bits and why

3.3.2 Branch Metric Calculation

number of bits, how are they stored

3.3.3 Branch Metric Storage

describe why the metrics were stored as they were,. to minimize read address generators

3.3.4 Path Metric Memory

3.3.5 Add-Compare-Select (ACS) Circuit

3.3.6 Optimal Path Memory

3.3.7 Optimal Path Memory Update Circuit

3.3.8 Decoder Control Circuits

3.3.8.1 Memory Address Generation Circuit

3.3.8.2 Master Control Circuit

3.3.8.3 Slave Control Circuits

3.4 Decoder Device Utilization

3.5 Decoder Performance

3.6 Expanding the Decoder to Support IS-2000

3.6.1 Branch Metric Circuit optimizations

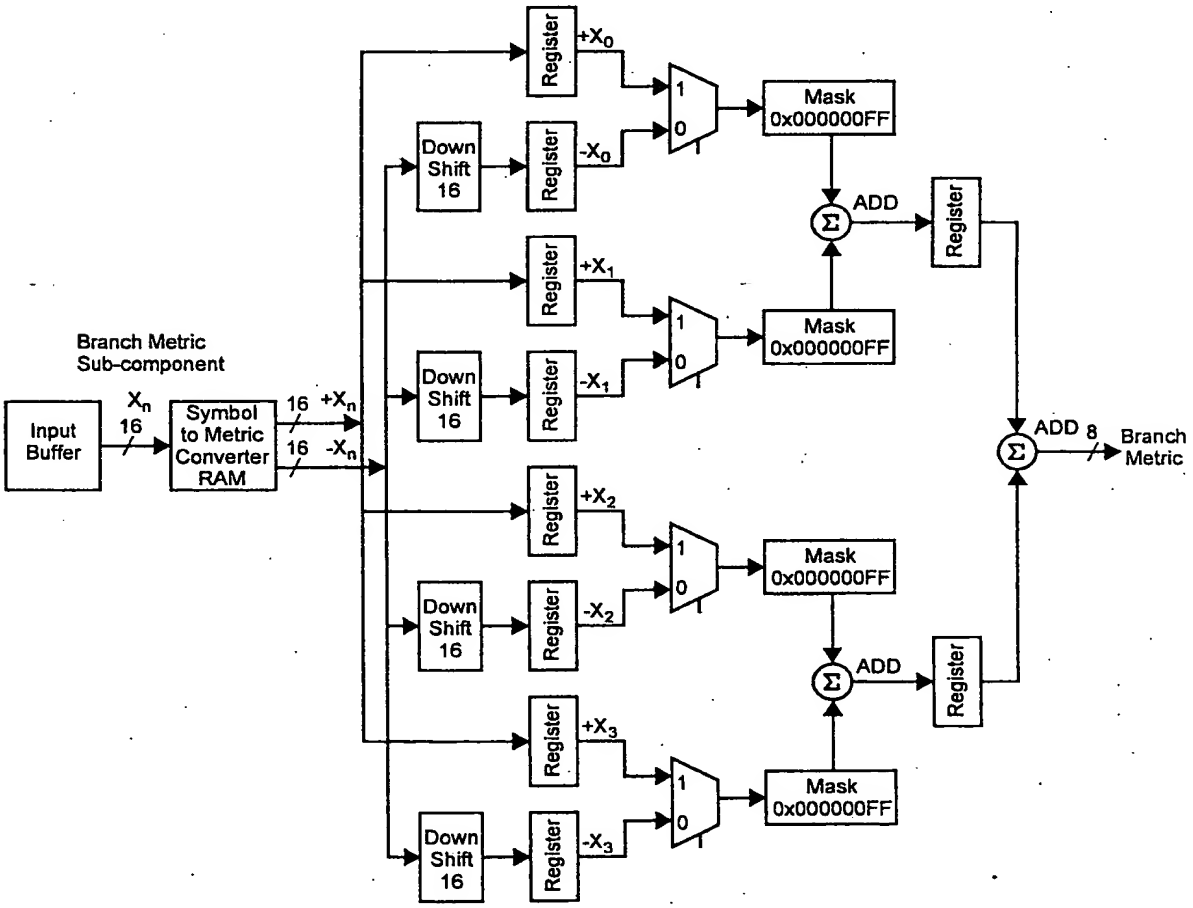
For each received symbol at time n, the branch metrics $BM_n(k,0)$ and $BM_n(k,1)$ are needed to compute the path metric for each of the $2^{K-1} = 256$ states. Since X_{mi} can only be either -1 or +1, we can note that there are only sixteen unique values of $BM_n(k,m)$. Rather than compute 512 redundant branch metrics, we store the sixteen unique branch metrics in LSM memories for later use.

To determine which branch metric equations are required at each Add, Compare, Select (ACS) butterfly circuit in the path metric circuits

In order to minimize on-chip resources in the Viterbi decoder's branch metric calculation circuit, we can exploit the generator polynomial equations. It can be seen

Branch Metric Circuit Implementation

Figure 3-1. Branch Metric Calculation Circuit



3.6.2 Path Metric Calculation

Figure 3-2. General ACS Circuit

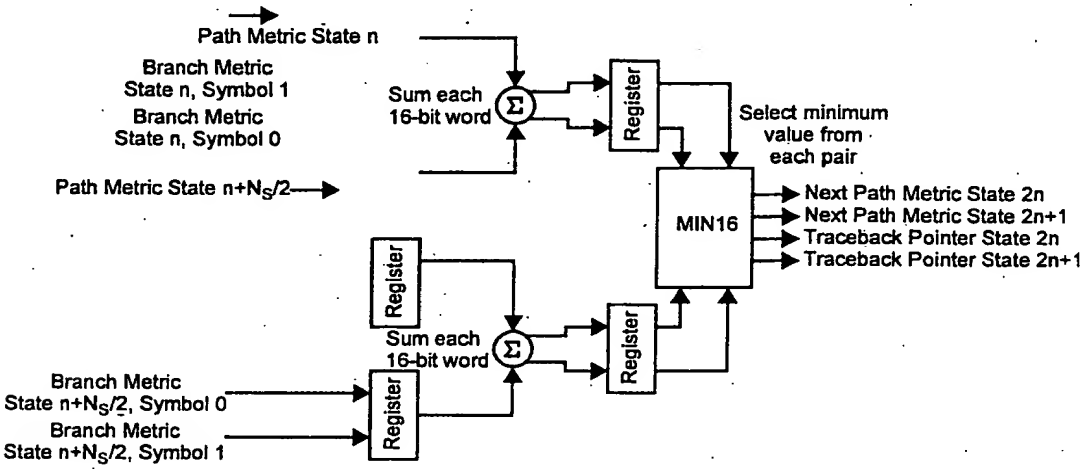
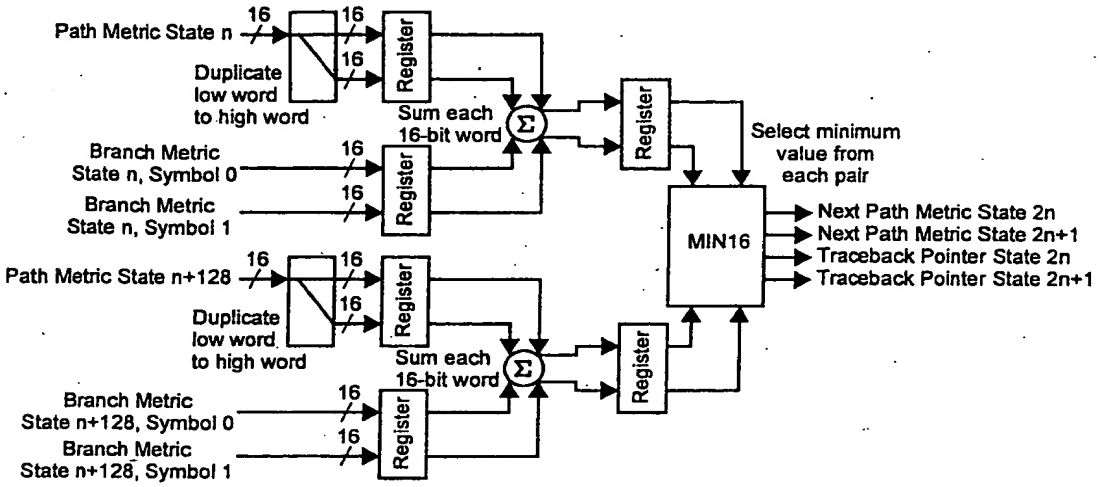


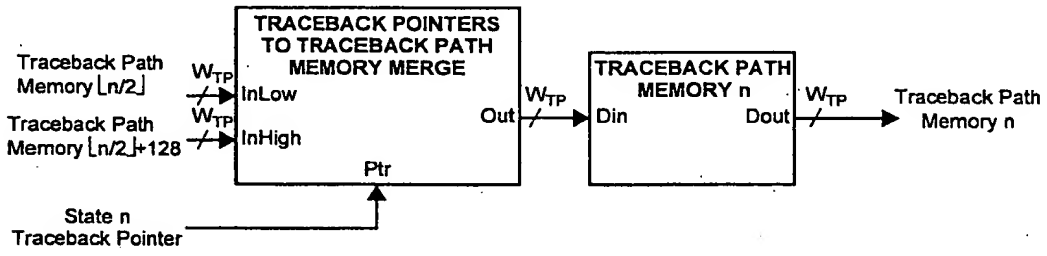
Figure 3-3. CS2000 Path Metric Update Circuit



3.6.3 Traceback Path Memory Update

The Traceback Path Memory (TPM) circuit assumes that the decoder has at least two Path Metric circuits operating in parallel. Recall that the best traceback path comes from one of only two for a rate 1/2, 1/3, or 1/4 convolutional encoder. The TPM Update circuit fetches the last WTP bits from the two possible states feeding

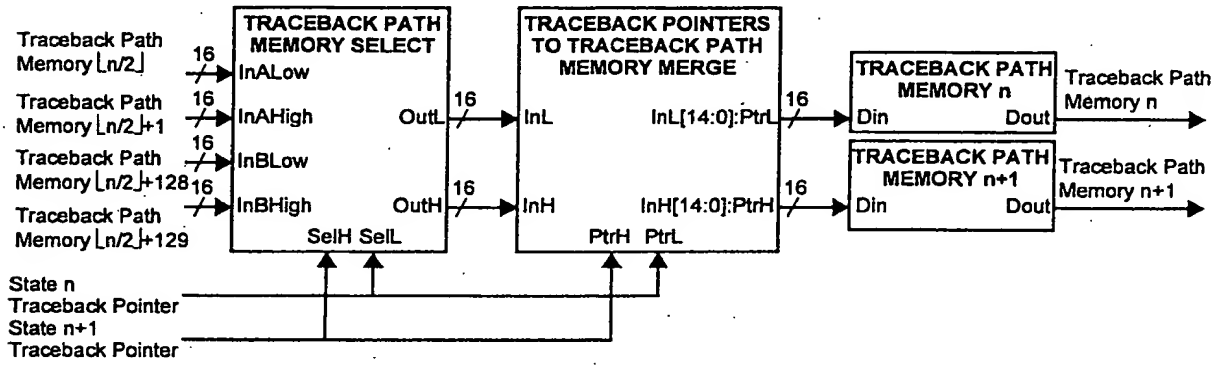
Figure 3-4. Single Traceback Path Memory Update Module



TRACEBACK PATH MEMORY SELECT TRUTH TABLE	
Sel	**Out**
0	Ptr:(TP Mem ⌊n/2⌋)[14:0]
1	Ptr:(TP Mem ⌊n/2⌋+128)[14:0]

Extra diagrams (to be deleted)

Figure 3-5. Dual Traceback Path Memory Update Module



TRACEBACK PATH MEMORY SELECT TRUTH TABLE			
SelH	SelL	OutH	OutL
0	0	TP Mem⌊n/2⌋+1	TP Mem⌊n/2⌋
0	1	TP Mem⌊n/2⌋+1	TP Mem⌊n/2⌋+128
1	0	TP Mem⌊n/2⌋+129	TP Mem⌊n/2⌋
1	1	TP Mem⌊n/2⌋+129	TP Mem⌊n/2⌋+128